DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Programación SIMD

Organización del Computador II

Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Agustina Aldasoro	86/13	agusaldasoro@gmail.com
Maximiliano Rey	37/13	rey.maximiliano@gmail.com
Ignacio Tirabasso	718/12	ignacio.tirabasso@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Resumen

En el presente trabajo se describen los beneficios de la programación en Lenguaje Ensamblador bajo el modelo de programación SIMD mediante el uso de instrucciones SSE.

Índice

1. Objetivos generales

El objetivo de este Trabajo Práctico es evaluar la eficiencia del modelo de programación SIMD mediante la implementación de diversos algoritmos en lenguaje Ensamblador utilizando instrucciones SSE.

Las mediciones se realizan mediante pruebas empíricas del código frente a algoritmos que cumplen la misma especificación, implementados en un lenguaje de alto nivel (C).

En este proyecto, los algoritmos a implementar se basaron en el procesamiento de imágenes y video, en el cual el uso del modelo SIMD es provechoso.

2. Contexto

Abordando el objetivo de este trabajo, realizamos una experimentación enfocada en reducir el tiempo de cómputo de un programa basado en dos sucesos que tienen la capacidad de afectarlo.

Por un lado se encuentra la *capacidad de cómputo*, la cual limita la cantidad de operaciones aritméticas que el procesador puede paralelizar. Si el programa ejecuta un uso intensivo en operaciones aritméticas, al añadirle nuevas operaciones de esta índole se van a necesitar más ciclos de clock para ejecutarlas.

El otro cuello de botella importante es el *ancho de banda de la memoria*. Cuando el programa ejecuta una instrucción que implica un acceso a memoria, se precisan más ciclos de clock para que la memoria responda, en particular si el dato no se encuentra en el cache.

Diseñamos nuestros casos de testeo con el fin de observar para cada programa cuál es el factor que determina el tiempo de cómputo.

Nuestra experimentación se centra en la cantidad de ciclos de clock que transcurren desde el inicio hasta el final de la ejecución del programa. Se adjunta con la documentación los archivos *.py utilizados para calcular la esperanza y la varianza de cada una de las mediciones.

Si lo vamos a rearmar, habria que explicar un poquito mas que hace el codigo, cuantas iteraciones y que descarta los maximos y minimos

3. Implementación en Assembler

3.1. Filtro CropFlip

Luego de pushear los cinco registros a utilizar, almacenamos:

```
Pushear la base de la pila y los registros a utilizar.
Alinear la pila.
mov r12d, [rbp+16] ;tamx
mov r13d, [rbp+24] ;tamy
mov r14d, [rbp+32] ;offsetx
mov r15d, [rbp+40] ;offsety
Limpiamos la parte alta de estos registros haciendo un mov rXd, rXd.
```

Utilizamos el registro **r10** como el *y-actual* (fuente) y el registro **r11** como el *x-actual* (fuente). Recorremos la imagen fuente desde arriba hacia abajo, de izquierda a derecha.

```
mov r10,r15 ; y
mov r11,r14 ; x
```

Utilizamos el registro **rcx** como el y_2*-actual* (destino) y el registro **rdx** como el x_2*-actual* (destino). Recorremos la imagen destino de abajo hacia arriba, de izquierda a derecha. Al registro rcx debemos decrementarlo en uno porque arranca inicializado en cero.

```
mov rcx,r13 ; y2 = tamy
dec rcx     ; y2 = tamy-1
mov rdx,0   ; x2 = 0
```

En **r13** almacenamos el límite para r10, es decir tiene que recorrer el ciclo de y hasta que alcance su límite en y (offsety+tamy).

```
add r13,r15 ; r13 = offsety+tamy
```

En **rbx** almacenamos el límite para r11, es decir tiene que recorrer el ciclo de x hasta que alcance su límite en x (offsetx+tamx)

```
mov rbx,r12  ; rbx = tamx
add rbx,r14  ; rbx = offsetx+tamx
```

Ahora comienza el ciclo de iteración sobre la variable *y*. Cada vez que se ejecuta se comprueba que el *y-actual* (**r10**) sea menor que su límite (**r13**). Y por cada iteración se reinician los valores de *x* e x_2 a la primer columna que debemos trabajar (Para la imagen fuente es el offset inicial de la variable *x*, para la imagen destino es el *0*).

```
.loop_y:
cmp r10,r13     ; y < offsety+tamy
jge .endloop_y
mov r11,r14     ; x = offsetx
mov rdx,0       ; x2 = 0
```

Por consiguiente, para cada valor que vaya tomando la variable, se debe ejecutar un ciclo para poder iterar sobre todas las columnas (Ciclo de x). Por cada iteración, se compara si el *x-actual* (**r11**) es menor

que su límite (**rbx**). Si no lo es, se salta fuera del ciclo de x.

Luego, comienza a ejecutar el código propio del ciclo. Es necesario tener en cuenta que:
⋆ En **r8** está cargado el *row_size* de la imagen fuente.
⋆ En **r9** está cargado el *row_size* de la imagen de destino.

Vamos a calcular en cada iteración la cantidad de posiciones en memoria (bytes) que se le deben sumar a la posición de origen de la imagen para encontrarnos en la posición actual. Esto se almacena en el registro **rax**. Primero copiamos el contenido de **r10** (*y-actual*), lo multiplicamos por el largo de cada fila para así posicionarnos en la fila actual y por último le sumamos **r11** (que indica el número de fila actual) multiplicado por 4 porque cada Pixel tiene 4 bytes. De este modo, **rax** contiene el offset que debemos sumarle a la posición en memoria de la imagen fuente para situarnos en la posición actual.

Como nos encontramos utilizando registros Xmm, vamos a trabajar con 4 pixels a la vez, de modo que entran en un solo registro. Para levantar de memoria 4 pixels, utilizamos la intrucción *movdqu*.

Por último, lo que tenemos que hacer es guardar esos 4 pixels levantados con el mismo orden en la posición correspondiente de la imagen destino. Es decir, la misma columna x pero sin su offset (por eso mismo se usan dos registros distintos, ya que el offset de la imagen de destino es 0) y la fila y_2 va a ser la diferencia entre la cantidad de filas e y.

Análogamente a lo anterior, en **rax** se guarda el offset de la imagen destino, para acceder a memoria sumándoselo a la posición donde esta almacenada la imagen.

Se suma 4 a las variables r11(x-*actual*) y rdx(x_2-*actual*), porque en este paso avanzamos 4 pixels. Y el jump se ejecuta siempre, ya que la comparación se produce al principio del ciclo.

```
.loop_x:

cmp r11,rbx      ; x < offsetx+tamx
jge .endloop_x

mov rax,r10
imul rax,r8
lea rax,[rax+r11*4]

movdqu xmm0,[rdi+rax]

mov rax,rcx
imul rax,r9
lea rax,[rax+rdx*4]

movdqu [rsi+rax],xmm0

add r11,4
add rdx,4
jmp .loop_x
```

Una vez copiada toda una fila, debemos avanzar hacia la siguiente. De este modo: incrementamos en uno r10(*y-actual*) y rcx(y_2-*actual*).

Se ejecuta siempre un jump hacia el comienzo del ciclo en y, porque la comparación de ver si ya copiamos todas las filas se ejecuta al principio.

```
.endloop_x:
inc r10
dec rcx
jmp .loop_y
```

Una vez terminado el ciclo de y, sólo resta salir de la ejecución respetando la convención C.

```
.endloop_y:
 Alinear la pila y popear todos los registros.
 ret
```

3.2. Filtro Sierpinsky

Hola

3.3. Filtro Bandas

Hola

3.4. Filtro Motion Blur

Primero hacemos algunos define que vamos a necesitar luego:

```
cerocomados  dd  0.2 , 0.2 , 0.2 , 0.2
rgb_only     dd  0x00FFFFFF, 0x00FFFFFF, 0x00FFFFFF, 0x00FFFFFF
```

```
Salvamos la base de la pila y los registros a utilizar. Alineamos la pila.
Limpiamos con xor o pxor los registros a utilizar: r10, r11, r15, r13 y xmm13.

; r10 = i
; r11 = j
mov r15d,edx ; r15 = cols
mov r14d,ecx ; r14 = filas
sub r15,2   ; r15 = cols -2
sub r14,2   ; r14 = filas - 2
```

El registro **xmm13** es limpiado para luego utilizarlo a la hora de asignarle el borde negro. Se les resta 2 a las filas y a las columnas porque los últimos dos píxeles del borde van de negro.

Comienza el ciclo de y, es decir, primero recorremos por filas y por cada fila recorremos por columnas. Se recorren todas las filas y cada vez que se entra en el *loop_x* se inicializa la fila en cero.

```
.loop_y:
    cmp r10d,ecx  ; i < filas
    jge .endloop_y

    mov r11,0  ; j = 0
```

El *loop_x* se recorre hasta que j alcanza el valor de la última columna. En el registro **rax** vamos a asignarle el número de orden del byte a tratar, es decir la fila actual multiplicada por la cantidad de columnas sumado a la columna actual multiplicada por cuatro porque es lo que ocupa un píxel. Antes de asignarle el nuevo valor a la imagen, debemos comprobar si el píxel a tratar es de borde en cuyo caso queda negro (.cero) o si no lo es (.nocero). Todos los píxeles que son bordes son los que están ubicados en las primeras dos filas o columnas y los que están ubicados en las últimas dos filas o columnas.

```
.loop_x:
    cmp r11d,edx ; j < cols
    jge .endloop_x

    mov rax,r10  ; rax = i
    imul eax,r8d ; rax = i * row_size
    lea rax,[rax+r11*4] ; rax = i * row_size + j * 4

    cmp r11,2
    jl .cero    ; j < 2
    cmp r10,2
    jl .cero    ; i < 2
    cmp r10,r14
    jge .cero   ; i >= cols -2
    cmp r11,r15
    jge .cero   ; j >= filas -2

    jmp .nocero
```

Si el píxel a tratar debe quedar en negro (cero) lo que debemos hacer es asignarle ceros mediante el registro **xmm13**:

```
.cero:
    movq [rsi+rax],xmm13
    add r11,2
    jmp .loop_x
```

A continuación comienza la rutina a llevar a cabo en caso de que el píxel a tratar no sea borde. Se copia en **rbx** el puntero actual con el que vamos a trabajar. Recordar que en **rax** teníamos el offset en la imagen y en **rdi** el puntero al comienzo de la misma. Vamos a ejecutar 4 píxeles por vez (porque son los que entran en un registro xmm) y para ello necesitamos hacer cinco lecturas a memoria y así tener para cada píxel sus cuatro vecinos que van a influir en su valor final. En **xmm14** copiamos el contenido de *rgb_only* el cual nos permitirá sólo trabajar con los canales RGB de cada píxel más adelante.

```
.nocero:

    lea rbx,[rdi+rax]
    movdqu xmm1,[rbx]        ; (i,j)
    movdqu xmm2,[rbx+r8*1+4] ; (i+1,j+1)
    movdqu xmm3,[rbx+r8*2+8] ; (i+2,j+2)
    mov r12,rbx
    sub r12,r8
    movdqu xmm4,[r12-4]      ; (i-1,j-1)
    sub r12,r8
    movdqu xmm5,[r12-8]      ; (i-2,j-1)

    movdqu xmm14,[rgb_only]
```

Para continuar necesitamos desempaquetar los valores para poder hacer las cuentas, ya que necesitamos los valores de los píxeles agrupados en números de mayor tamaño para el cual contemos con sus instrucciones necesarias y también un orden de precisión mayor. Para ello, los expandimos con ceros que van a provenir de haber limpiado el registro **xmm0**.

```
    pxor xmm0,xmm0
    movdqu xmm6,xmm1      ; xmm6 = (i,j) || (i,j+1) || (i,j+2) || (i,j+3)
    punpcklbw xmm1,xmm0   ; xmm1 = (i,j) || (i,j+1)
    punpckhbw xmm6,xmm0   ; xmm6 = (i,j+2) || (i,j+3)

    movdqu xmm7,xmm2      ; xmm7 = (i+1,j+1) || (i+1,j+2) || (i+1,j+3) || (i+1,j+4)
    punpcklbw xmm2,xmm0   ; xmm2 = (i+1,j+1) || (i+1,j+2)
    punpckhbw xmm7,xmm0   ; xmm7 = (i+1,j+3) || (i+1,j+4)

    movdqu xmm8,xmm3      ; xmm8 = (i+2,j+2) || (i+2,j+3) || (i+2,j+4) || (i+2,j+5)
    punpcklbw xmm3,xmm0   ; xmm3 = (i+2,j+2) || (i+2,j+3)
    punpckhbw xmm8,xmm0   ; xmm8 = (i+2,j+4) || (i+2,j+5)

    movdqu xmm9,xmm4      ; xmm9 = (i-1,j-1) || (i-1,j) || (i-1,j+1) || (i-1,j+2)
    punpcklbw xmm4,xmm0   ; xmm4 = (i-1,j-1) || (i-1,j)
    punpckhbw xmm9,xmm0   ; xmm9 = (i-1,j+1) || (i-1,j+2)

    movdqu xmm10,xmm5     ; xmm10 = (i-2,j-1) || (i-2,j) || (i-2,j+1) || (i-2,j+2)
    punpcklbw xmm5,xmm0   ; xmm5 = (i-2,j-1) || (i-2,j)
    punpckhbw xmm10,xmm0  ; xmm10 = (i-2,j+1) || (i-2,j+2)
```

Por cada registro tenemos dos píxeles, de este modo ya podemos realizar la suma entre los cinco píxeles vecinos que luego de dividirlo por la constante van a ser asignados al píxel central de estos cinco. De este modo, contamos con la instrucción *paddw* la cual suma empaquetadamente de a Word (2 bytes). Utilizamos los registros **xmm1** y **xmm6** como acumuladores de la suma.

```
paddw xmm1,xmm2     ; xmm1 = (i,j)+(i+1,j+1)  ||  (i,j+1)+(i+1,j+2)
paddw xmm1,xmm3     ; xmm1 = (i,j)+(i+1,j+1)+(i+2,j+2)  ||  (i,j+1)+(i+1,j+2)+(i+2,j+3)
paddw xmm1,xmm4     ; xmm1 = (i,j)+(i+1,j+1)+(i+2,j+2)+(i-1,j-1)  ||
                             (i,j+1)+(i+1,j+2)+(i+2,j+3)+(i-1,j)
paddw xmm1,xmm5     ; xmm1 = (i,j)+(i+1,j+1)+(i+2,j+2)+(i-1,j-1)+(i-2,j-1)  ||
                             (i,j+1)+(i+1,j+2)+(i+2,j+3)+(i-1,j)+(i-2,j)

paddw xmm6,xmm7     ; xmm6 = (i,j+2)+(i+1,j+3)  ||  (i,j+3)+(i+1,j+4)
paddw xmm6,xmm8     ; xmm6 = (i,j+2)+(i+1,j+3)+(i+2,j+4)  ||  (i,j+3)+(i+1,j+4)+(i+2,j+5)
paddw xmm6,xmm9     ; xmm6 = (i,j+2)+(i+1,j+3)+(i+2,j+4)+(i-1,j+1)  ||
                             (i,j+3)+(i+1,j+4)+(i+2,j+5)+(i-1,j+2)
paddw xmm6,xmm10    ; xmm6 = (i,j+2)+(i+1,j+3)+(i+2,j+4)+(i-1,j+1)+(i-2,j+1)  ||
                             (i,j+3)+(i+1,j+4)+(i+2,j+5)+(i-1,j+2)+(i-2,j+2)
```

Para poder multiplicarlos por una constante debemos transformar los valores en puntos flotantes, para ello primero debemos desempaquetarlos ampliandolos con cero (adquiriendo así un tamaño de Doubleword) y luego mediante la instrucción *cvtdq2ps* convertirlos de enteros con signo a puntos flotantes de precisión simple sin perder el empaquetamiento que nos permite trabajar con los tres valores del píxel a la vez. (En realidad son cuatro, pero un canal no lo usamos).

```
movdqu xmm2,xmm1        ; xmm2 = (i,j)+(i+1,j+1)+(i+2,j+2)+(i-1,j-1)+(i-2,j-1)   ||
                                 (i,j+1)+(i+1,j+2)+(i+2,j+3)+(i-1,j)+(i-2,j)
movdqu xmm7,xmm6        ; xmm7 = (i,j+2)+(i+1,j+3)+(i+2,j+4)+(i-1,j+1)+(i-2,j+1) ||
                                 (i,j+3)+(i+1,j+4)+(i+2,j+5)+(i-1,j+2)+(i-2,j+2)

punpcklwd xmm1,xmm0     ; xmm1 = (i,j)+(i+1,j+1)+(i+2,j+2)+(i-1,j-1)+(i-2,j-1)
punpckhwd xmm2,xmm0     ; xmm2 = (i,j+1)+(i+1,j+2)+(i+2,j+3)+(i-1,j)+(i-2,j)

punpcklwd xmm6,xmm0     ; xmm6 = (i,j+2)+(i+1,j+3)+(i+2,j+4)+(i-1,j+1)+(i-2,j+1)
punpckhwd xmm7,xmm0     ; xmm7 = (i,j+3)+(i+1,j+4)+(i+2,j+5)+(i-1,j+2)+(i-2,j+2)

cvtdq2ps xmm1,xmm1
cvtdq2ps xmm2,xmm2
cvtdq2ps xmm6,xmm6
cvtdq2ps xmm7,xmm7
```

Ahora que los valores ya están trasnformados en puntos flotantes, podemos hacer la multiplicación (también empaquetada). De este modo, calculamos todo un píxel por cada registro. Una vez hechas las multiplicaciones, volvemos a convertir los valores en enteros mediante la instrucción *cvtps2dq*. Como habíamos definido antes, *cerocomados* guarda el valor 0.2 cuatro veces.

```
movdqu xmm15,[cerocomados]

mulps xmm1,xmm15
mulps xmm2,xmm15
mulps xmm6,xmm15
mulps xmm7,xmm15

cvtps2dq xmm1,xmm1
cvtps2dq xmm2,xmm2
cvtps2dq xmm6,xmm6
cvtps2dq xmm7,xmm7
```

Ahora resta volver a empaquetar para que queden los cuatro píxeles, que estábamos trabajando en simultaneo, en un sólo registro. Primero juntamos el valor final de **xmm1** (i, j) con el de **xmm2** $(i, j+1)$ y el de **xmm6** $(i, j+2)$ con el del **xmm7** $(i, j+3)$ y por último unimos todos en un mismo registro (**xxm1**). Guardamos el contenido de xmm1 en memoria, a la dirección calculada previamente en **rax**, sumada al origen del puntero destino (**rsi**). Sumamos cuatro al contador de **r11** (columnas) porque por cada lectura en memoria, trabajamos con 4 píxeles, y luego volvemos al ciclo de x.

```
    packusdw xmm1,xmm2
    packusdw xmm6,xmm7

    packuswb xmm1,xmm6

    movdqu [rsi+rax],xmm1

    add r11,4
    jmp .loop_x
```

Al terminar el ciclo de x, lo que hacemos en incrementar en uno **r10**, el cual mide la fila actual. Es decir, avanzamos de fila y saltamos al ciclo de y.

```
.endloop_x:
    inc r10
    jmp .loop_y
```

Al terminar el ciclo de y, queda finalizada la ejecución del filtro. Por lo tanto, solo queda restaurar los registros y retornar.

```
.endloop_y:
    Hacemos pop de los registros salvados
    ret
```

4. Enunciado y solución

4.1. Mediciones

Realizar una medición de performance *rigurosa* es más difícil de lo que parece. En este experimento deberá realizar distintas mediciones de performance para verificar que sean buenas mediciones.

En un sistema "ideal" el proceso medido corre solo, sin ninguna interferencia de agentes externos. Sin embargo, una PC no es un sistema ideal. Nuestro proceso corre junto con decenas de otros, tanto de usuarios como del sistema operativo que compiten por el uso de la CPU. Esto implica que al realizar mediciones aparezcan "ruidos" o "interferencias" que distorsionen los resultados.

El primer paso para tener una idea de si la medición es buena o no, es tomar varias muestras. Es decir, repetir la misma medición varias veces. Luego de eso, es conveniente descartar los outliers [1], que son los valores que más se alejan del promedio. Con los valores de las mediciones resultantes se puede calcular el promedio y también la varianza, que es algo similar el promedio de las distancias al promedio[2].

Las fórmulas para calcular el promedio μ y la varianza σ^2 son

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \sigma^2 = \frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}$$

4.2. Filtro *cropflip*

Programar el filtro *cropflip* en lenguaje C y luego en ASM haciendo uso de las instrucciones vectoriales (**SSE**).

Experimento 1.1 - análisis el código generado

En este experimento vamos a utilizar la herramienta `objdump` para verificar como el compilador de C deja ensamblado el código C.

Ejecutar

```
objdump -Mintel -D cropflip_c.o
```

¿Cómo es el código generado? Indicar *a*) Por qué cree que hay otras funciones además de `cropflip_c` *b*) Cómo se manipulan las variables locales *c*) Si le parece que ese código generado podría optimizarse

Muchas de las funciones generadas se agregan al compilar usando parámetros de debugging.

El cambio que puede notarse a simple vista es que el tamaño del código generado con optimización es considerablemente menor al generado sin optimizaciones, un 34 % menor. Por otro lado, calcula una única vez la posición de la cual deberá leer el pixel a procesar, ya que es una operación que se realiza cuatro veces por cada lectura y escritura de un pixel.

El código optimizado también disminuye el uso del stack para variables locales y en su lugar utiliza registros, por esa razón necesita armar el stackframe y pushear los registros que la convención C pide preservar. Esto no se da en el código sin optimizar, todas las variables locales se guardar en el stack, se asignan a algún registro libre al momento de usarla y luego vuelve a ser guardada en el stack. Si bien la cantidad de accesos a memoria es sustencialmente mayor en el código sin optimizar por el hecho de que se accede constantemente a las variables locales alojadas en el stack, también hay que considerar que al haber tantos accesos al stack el hit-rate de la caché debería ser bastante alto, agilizando la lectura, no así la escritura.

Experimento 1.2 - optimizaciones del compilador

[1] en español, valor atípico: `http://es.wikipedia.org/wiki/Valor_atpico`

[2] en realidad, elevadas al cuadrado en vez de tomar el módulo

Compile el código de C con flags de optimización. Por ejemplo, pasando el flag `-O1`[3]. Indicar 1. Qué optimizaciones observa que realizó el compilador 2. Qué otros flags de optimización brinda el compilador 3. Los nombres de tres optimizaciones que realizan los compiladores.

GCC provee un arsenal de optimizaciones disponibles para que podamos usar cuando lo creamos conveniente, asímismo provee unos flags para poder compilar el código con un conjunto de optimizaciones, estos son O1, O2, O3, Os, entre otros. O1, O2 y O3 son optimizaciones generales no agresivas, es decir, no deberían modificar el funcionamiento del programa. Otras optimizaciones podrían asumir que todas las operaciones aritméticas son sin signo o que están bien implementadas y no es necesario chequear la consistencia de los resultados (fno-math-errno). Estas optimizaciones tienen como objetivo mejorar la performance del programa, sin embargo Os tiene como objetivo reducir el tamaño del ejecutable, es decir, activa todas las optimizaciones que contribuyen a reducir la cantidad de instrucciones regardless del impacto que esto pueda tener en la performance del programa.

Concretamente al activar las optimizaciones O3 en cropflip el cambio más notorio es que GCC utiliza instrucciones SSE para procesar 4 pixels por iteración, lo cual le brinda un boost de velocidad impresionante. Además de haberse reducido la cantidad de instrucciones.

Experimento 1.3 - calidad de las mediciones

1. Medir el tiempo de ejecución de cropflip 10 veces. Calcular el promedio y la varianza. Consideraremos outliers a los 2 mayores tiempos de ejecución de la medicion y también a los 2 menores, por lo que los descartaremos. Recalcular el promedio y la varianza después de hacer este descarte. Realizar un gráfico que presente estos dos últimos items.

   Luego de ejecutar 10 veces el filtro Cropflip obtuvimos los siguientes resultados:

ASM	**C**
70.925	1.152.187
70.521	1.151.544
32.859	761.937
43.720	649.248
64.236	1.152.847
70.793	1.153.061
71.271	1.152.765
44.616	1.152.798
56.124	725.420
71.775	1.155.718
Esperanza	
59.684	1.020.752,5
Desvío standard	
13.720,5329	203.626,443

   Aca puso Matias que el desvio standar en c no le da, VOLVER A CALCULARLO.
   El cuadro denota la cantidad de ciclos de clock utilizada por cada ejecución del programa.

   Luego de eliminar los dos valores más altos y los dos valores más bajos, recalculamos obteniendo los siguientes datos:
   Esperanza: 62.869,166 (ASM) y 1.087.346,333(C)
   Desvío standard: 9.719,205 (ASM) y 145.528,202(C)
   Se puede ver que al eliminar los outliers, la esperanza comienza a converger a su valor esperado y el desvío standard disminuye.

[3] agregando este flag a `CCFLAGS64` en el makefile

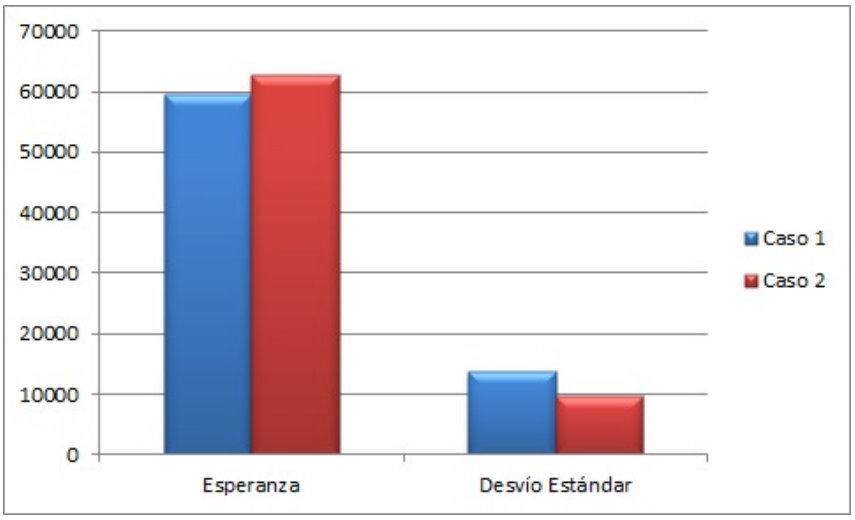


Figura 1: Assembler

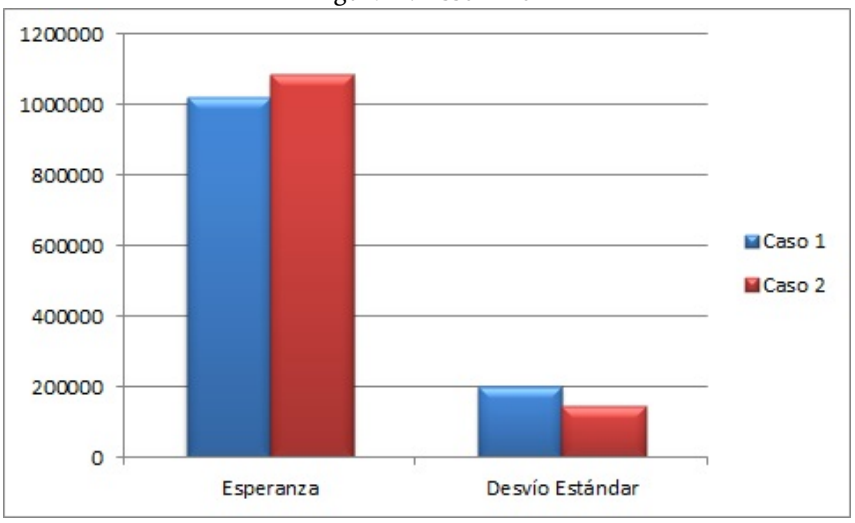


Figura 2: C

Siendo el Caso 1 las mediciones de esperanza y Desvío standard para todos los casos de test y el Caso 2 las mediciones sin tener en cuenta los cuatro outliers.

2. Implementar un programa en C que no haga más que ciclar infinitamente sumando 1 a una variable. Lanzar este programa tantas veces como *cores lógicos* tenga su procesador. Medir otras 10 veces mientras estos programas corren de fondo. Realizar los mismos casos de experimentación que en el ejercicio anterior.

Los resultados obtenidos en esta experimentación fueron menores que los anteriores:

ASM	C
33.585	542.928
33.798	544.155
33.402	544.857
33.228	543.687
33.159	543.252
33.441	543.324
34.089	544.224
33.768	760.359
34.563	542.448
34.473	542.982
Esperanza	
33.750,6	565.221,6
Desvío standard	
465,49	65.049,34

Luego de eliminar los dos valores más altos y los dos valores más bajos, recalculamos obteniendo los siguientes datos:
Esperanza: 33.680,5 (ASM) y 543.604(C)
Desvío standard: 235,364 (ASM) y 462,615(C)
Acá también se puede apreciar que al eliminar los outliers, el Desvío standard disminuye su valor.

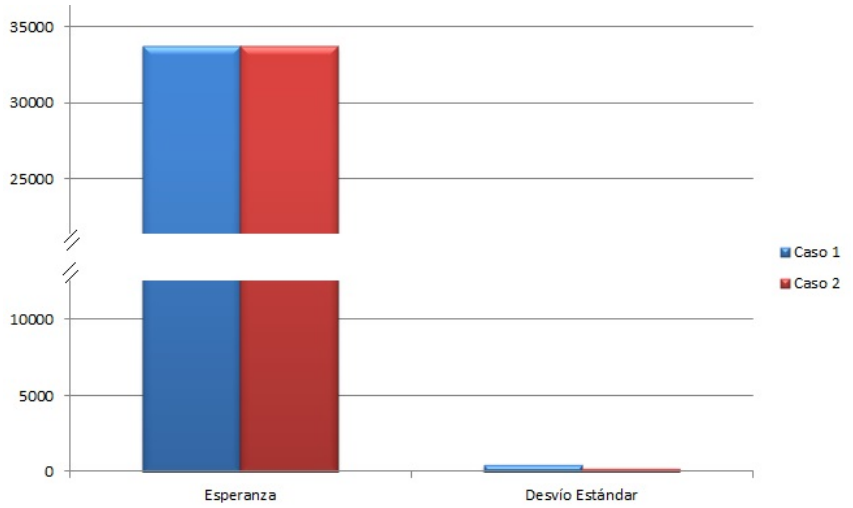


Figura 3: Assembler

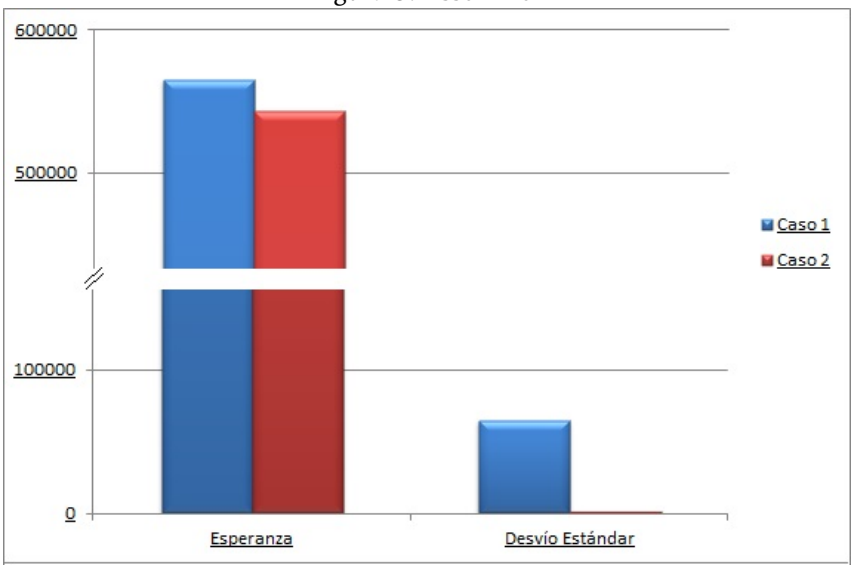


Figura 4: C

Siendo el Caso 1 las mediciones de esperanza y Desvío standard para todos los casos de test y el Caso 2 las mediciones sin tener en cuenta los cuatro outliers. Se puede observar que las mediciones mejoran con la ejecución del ciclo infinito de fondo, esto se debe a que fue ejecutado en una computadora con un procesador i5. Podria explicarse un poco mas que pasa con los is

Por este motivo, volvimos a ejecutar este caso una mayor cantidad de veces para que el procesador no modifique la frecuencia de clock, obteniendo los siguientes resultados:
Esperanza: 269.945,590 (ASM) y 9.524.152,001 (C)
Desvío Standard: 3.801,837 y 7.774.736,417 (C)
El grafico mas importante es este, HACERLO Y SACAR EL OTRO.
A partir de aquí todos los experimentos de mediciones deberán hacerse igual que en el presente ejercicio: tomando 10 mediciones, luego descartando outliers y finalmente calculando promedio y Desvío standard.

Decidimos:
Realizar 12000 mediciones por experimento, eliminando los primeros mil casos que hayan llevado menos ciclos de clock y los mil casos que hayan llevado la mayor cantidad de ciclos de clock.
Lo determinamos de esta manera, ya que dejar dos mediciones afuera, como dice el enunciado, no tiene

influencia en los cálculos de la esperanza y la varianza para muestras tan grandes.
Luego de experimentar distintas cantidades de casos de testeo, notamos que elegir 7000 valores pertenecientes a la franja del medio de los 12000 es una solución lo suficientemente estable, por lo cual es la que llevamos a cabo.

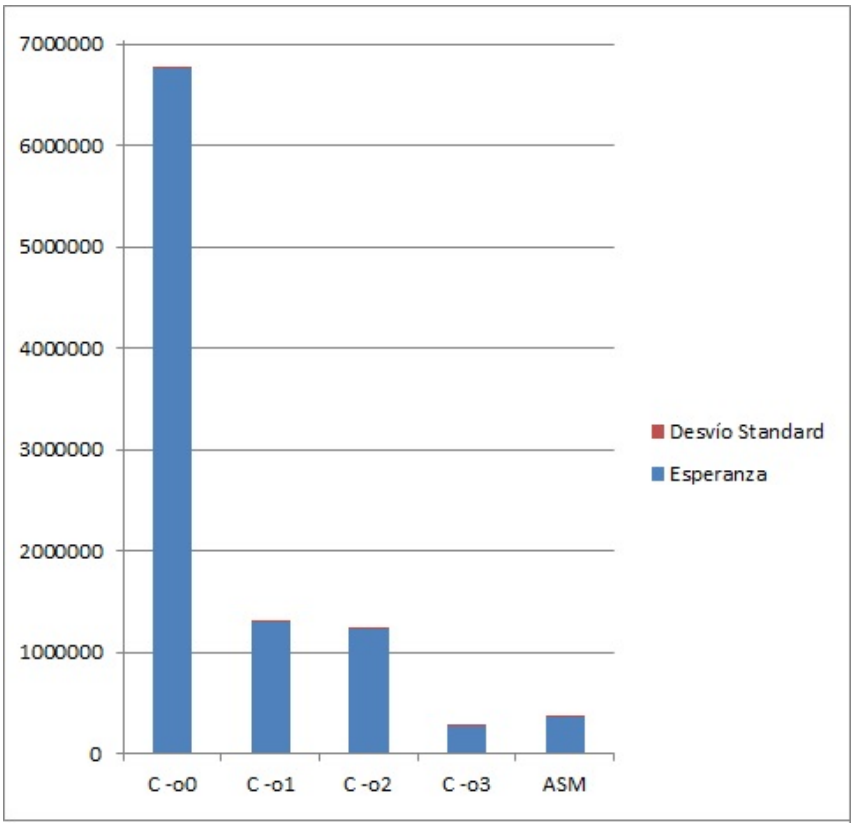
Experimento 1.4 - secuencial vs. vectorial

En este experimento deberá realizar una medición de las diferencias de performance entre las versiones de C y ASM (el primero con -O0, -O1, -O2 y -O3) y graficar los resultados.

El siguiente gráfico indica la esperanza de la cantidad de ciclos de clock que toma ejecutar el filtro Cropflip con los parámetros 404 404 4 4 en ASM y en C variando los flags de o0 a o3.
Reflejando las siguientes magnitudes:

	Esperanza	Desvío Standard
C -o0	6.761.044,5	131,463
C -o1	1.300.109	53,447
C -o2	1.227.102,5	50,143
C -o3	266.688	0,287
ASM	362.232	2,079

Se puede observar que la varianza es casi despreciable considerando el valor de la esperanza. Además, es notorio cómo el correr el programa con la orden de -o0 no efectúa ninguna optimización. También se puede observar que el código corrido en C bajo el comando de -o3 tiene una esperanza menor a la del código Assembler.



Experimento 1.5 - cpu vs. bus de memoria

Se desea conocer cual es el mayor limitante a la performance de este filtro en su versión ASM.

¿Cuál es el factor que limita la performance en este caso? En caso de que el limitante fuera la intensidad de cómputo, entonces podrían agregarse instrucciones que realicen accesos a memoria extra y la performance casi no debería sufrir. La inversa puede aplicarse, si el limitante es la cantidad de accesos a memoria. [4]

Realizar un experimento, agregando 4, 8 y 16 instrucciones aritméticas (por ej `add rax, rbx`) analizando como varía el tiempo de ejecución. Hacer lo mismo ahora con instrucciones de acceso a memoria, haciendo mitad lecturas y mitad escrituras (por ejemplo, agregando dos `mov rax, [rsp]` y dos `mov [rsp+8], rax`).[5]

Realizar un único gráfico que compare: 1. La versión original 2. Las versiones con más instrucciones aritméticas 3. Las versiones com más accesos a memoria Poner que instrucciones usamos!!!
Acompañar al gráfico con una tabla que indique los valores graficados.

Cropflip	Esperanza	Desvío Standard
Versión común	157.236,897	5.334,414
Con 4 instrucciones aritméticas	183.968,425	8.391,506
Con 8 instrucciones aritméticas	225.991,244	7.729,428
Con 16 instrucciones aritméticas	707.857,067	12.756,322
Con 4 accesos a memoria	281.032,176	11.258,907
Con 8 accesos a memoria	352.370,777	14.353,789
Con 16 accesos a memoria	342.023,334	12.136,799

[4] también podría pasar que estén más bien balanceados y que agregar cualquier tipo de instrucción afecte sensiblemente la performance

[5] Notar que en el caso de acceder a `[rbp]` o `[rsp+8]` probablemente haya siempre hits en la cache, por lo que la medición no será de buena calidad. Si se le ocurre la manera, realizar accesos a otras direcciones alternativas.

Se puede percibir cómo se ve afectado el tiempo de ejecución frente al incremento de instrucciones aritméticas más que por los accesos a Memoria.

Esta conclusion no se desprende inmediatamente del grafico. Con 4 y 8 accesos a memoria, el programa es más lento. Con 16 accesos pasa algo no intuitivo, que al menos debería mencionarse (y para intentar entenderlo es clave que pongan qué accesos usaron).
El pico de 16 instrucciones aritméticas tampoco me resulta intuitivo a simple vista.

4.3. Filtro *Sierpinski*

Programar el filtro *Sierpinski* en lenguaje C y en en ASM haciendo uso de las instrucciones vectoriales (**SSE**).

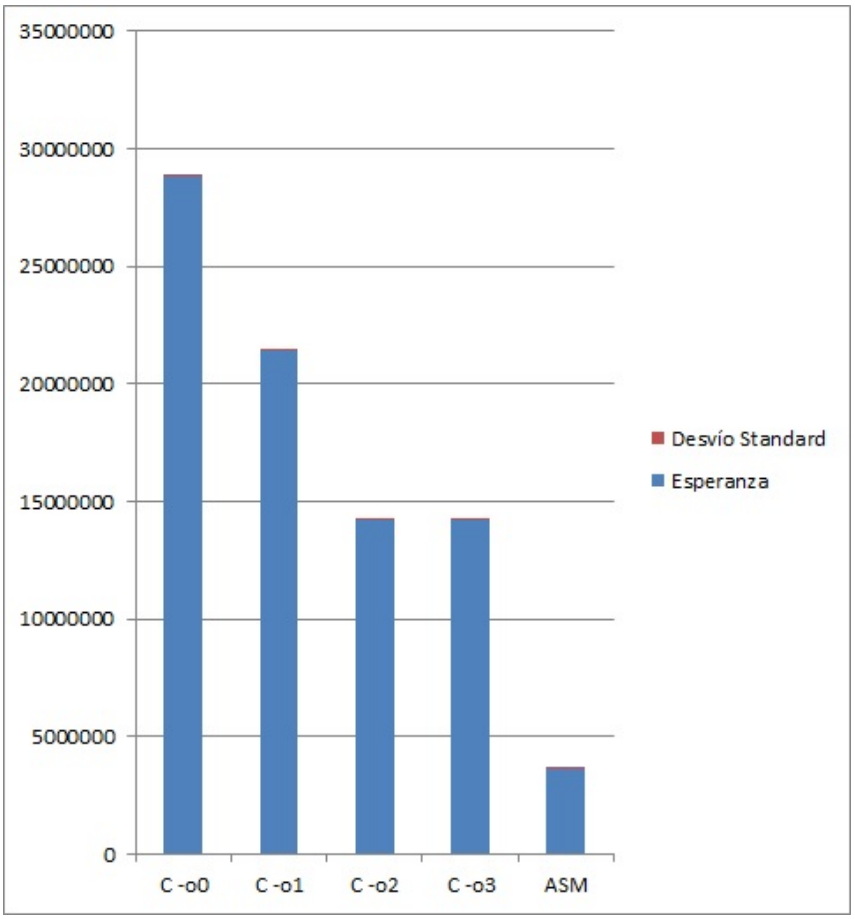
Experimento 2.1 - secuencial vs. vectorial

Analizar cuales son las diferencias de performace entre las versiones de C y ASM de este filtro, de igual modo que para el experimento 1.4.

El siguiente gráfico indica la esperanza de la cantidad de ciclos de clock que toma ejecutar el filtro Sierpinski en ASM y en C variando los flags de o0 a o3.
Reflejando las siguientes magnitudes:

	Esperanza	Desvío Standard
C -o0	28.801.586,5	78,016
C -o1	21.395.732	78,447
C -o2	14.231.758	116,445
C -o3	14.229.650	116,022
ASM	3.661.626	115,583

Aquí también el valor del desvío standard es despreciable. Donde correr el código en C bajo el comando -o0 sigue siendo el caso con mayor esperanza y además la esperanza del código Assembler es menor a la del código C con -o3.



Experimento 2.1 - cpu vs. bus de memoria

¿Cuál es el factor que limita la performance en este filtro? Repetir el experimento 1.5 para este filtro.

	Esperanza	Desvío Standard
Versión común	2.936.302,152	26.416,331
Con 4 instrucciones aritméticas	2.961.773,596	47.749,307
Con 8 instrucciones aritméticas	2.940.553,343	59.912,727
Con 16 instrucciones aritméticas	3.031.280,298	37.499,415
Con 4 accesos a memoria	2.919.253,679	57.697,386
Con 8 accesos a memoria	2.936.432,138	56.870,171
Con 16 accesos a memoria	3.051.926,505	40.964,812

En este caso, los accesos a memoria influyen más notoriamente la esperanza que las operaciones aritméticas lógicas.

Algo parecido al experimento 1.5
Además se podría mencionar que el desvío standard es elevado.

4.4. Filtro *Bandas*

Programar el filtro *Bandas* en lenguaje C y en en ASM haciendo uso de las instrucciones vectoriales (**SSE**).

Experimento 3.1 - saltos condicionales

Se desea conocer que tanto impactan los saltos condicionales en el código de filtro Bandas con `-O1` (la versión en C).
Para poder medir esto de manera aproximada, remover el código que detecta a que banda pertenece cada pixel, dejando sólo una banda. Por más que la imagen resultante no sea correcta, será posible tomar una medida aproximada del impacto de los saltos condicionales. Analizar como varía la performance.

En la siguiente figura se ve cómo varía la esperanza y el desvío standard entre dos corridas de C con el flag -o1 ambas:

En el gráfico anterior se puede ver que la influencia de los saltos condicionables es notable.

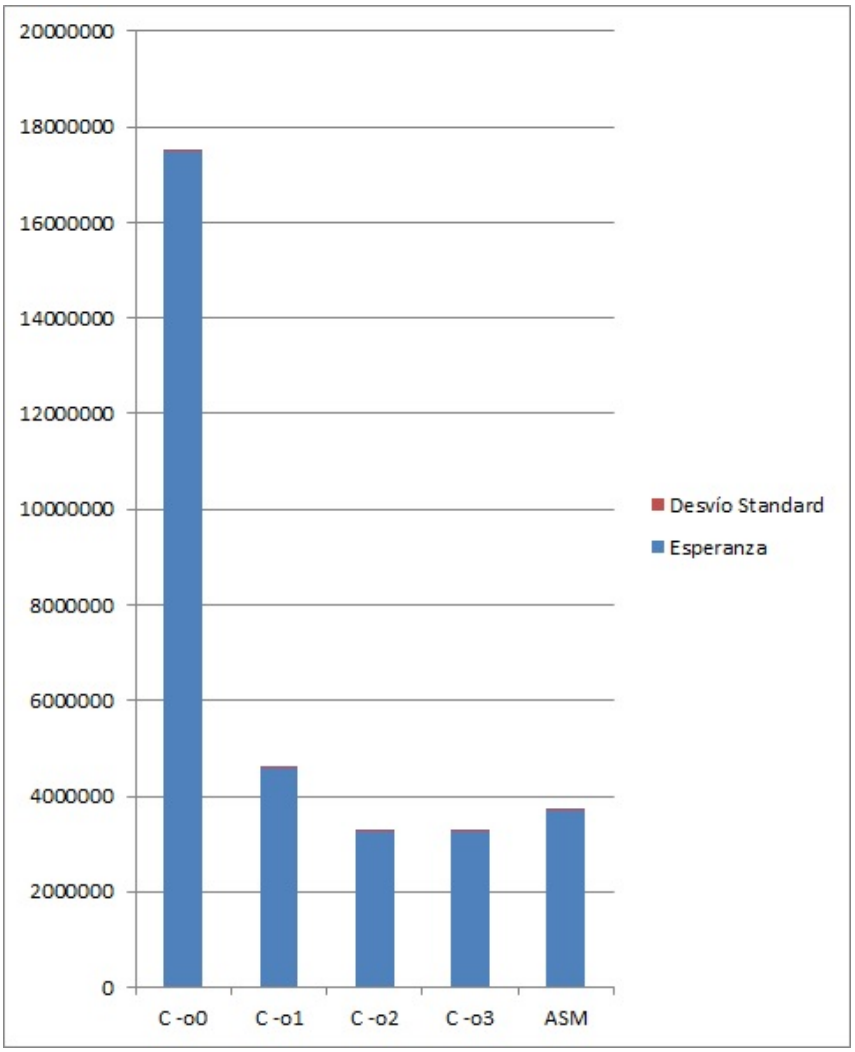
Experimento 3.2 - secuencial vs. vectorial

Repetir el experimento 1.4 para este filtro.

El siguiente gráfico indica la esperanza de la cantidad de ciclos de clock que toma ejecutar el filtro Bandas en ASM y en C variando los flags de o0 a o3.
Reflejando las siguientes magnitudes:

	Esperanza	Desvío Standard
C -o0	17.472.791	120,620
C -o1	4.583.340	128,514
C -o2	3.259.839	117,746
C -o3	3.259.767	117,391
ASM	3.703.203,5	115,659

Aquí también el valor del desvío standard es despreciable. En este caso, el tiempo de ejecución del código con el comando de -o3 tiene una esperanza menor a la del código Assembler. y -o2 tambien, por que?



4.5. Filtro *Motion Blur*

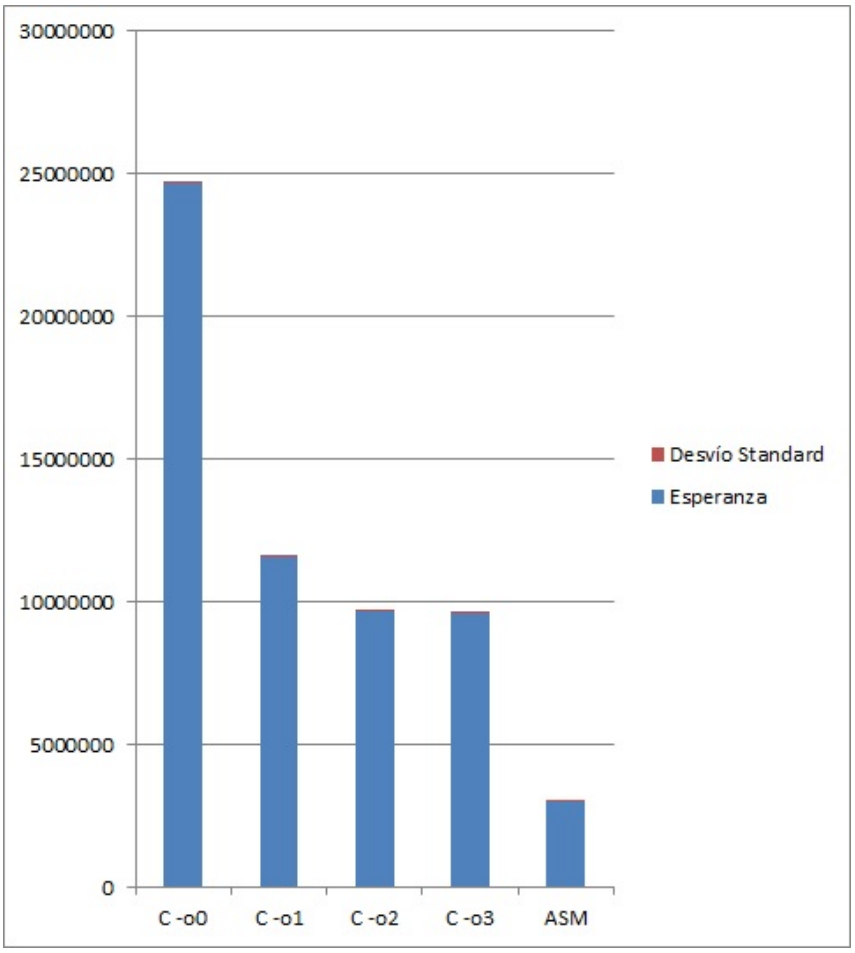
Programar el filtro *mblur* en lenguaje C y en ASM haciendo uso de las instrucciones **SSE**.

Experimento 4.1
Repetir el experimento 1.4 para este filtro

El siguiente gráfico indica la esperanza de la cantidad de ciclos de clock que toma ejecutar el filtro Bandas en ASM y en C variando los flags de o0 a o3.
Reflejando las siguientes magnitudes:

	Esperanza	Desvío Standard
C -o0	24.652.732,5	117,738
C -o1	11.562.261	103,024
C -o2	9.641.346	78,159
C -o3	9.639.625	78,227
ASM	2.993.980,5	118,245

Aquí también el valor del desvío standard es despreciable. Donde las optimizaciones del compilador en la versión -o3 no son suficientes para tener una esperanza menor a la esperanza obtenida bajo el código Assembler.



5. Conclusiones y trabajo futuro

Si bien un lenguaje ensamblador solo es utilizable en determinada gama de procesadores (ESTO HAY QUE SACARLO DIJO MATIAS), los resultados de nuestras mediciones han demostrado que la implementación de los filtros en este lenguaje han logrado una performance que el gcc no logro conseguir.

Dada la popularidad de la arquitectura de 64 bits de intel, se puede concluir que la implementación del SIMD en assembler tiene un precio razonable frente a la perfomance alcanzada.